On the circular path from RAII to crazy-town back to RAII: Thoughts on emulating C#'s using in C++

devblogs.microsoft.com/oldnewthing/20170710-00

July 10, 2017

Raymond Chen

Some follow-up notes on Emulating the C# `using` keyword in C++, primarily for the benefit of people from reddit who stumbled into the series and didn't understand the context of the discussion, because this was really part 6 of the series begun the previous week, even though it wasn't labeled as such. (And the title itself was a party trick rather than a serious proposal.)

The main complication that prevented us from using RAII was the use of the Parallel Patterns Library (PPL) to express asynchronous programming in C++.[1] The most general pattern for asynchronous programming is that you start an operation, and then specify a callback to be invoked when the operation completes. In traditional C-style programming, this callback is a boring function pointer, coupled with some reference data so the callback has context for why it is being called back. C++ provides lambdas which let you express the continuation as a callback *object*, which is much more convenient since you can express the control flow inline instead of having tiny pieces of control flow scattered all over your program. And lambda capture makes it easy to express what pieces of information needs to be carried forward to the continuation.

If we didn't have any asynchronous operations, then a basic RAII class would do the trick: When the RAII class destructs, the cleanup operation occurs. (In our example, the cleanup operation is calling `Close`.) The difficulty in the asynchonous case is that it is cumbersome to keep carrying this RAII class forward and preventing it from destructing until the entire chain of continuations has completed. That's where the `ensure_close` and `shared_close` classes entered the picture. But you still had to remember to carry them forward.

The magical step was the introduction of the not-yet-standard-but-hopefully-soon `co_await` keyword. This transforms the function into a state machine, where each `co_await` represents the end of a state. The current execution state is saved, and execution of the task suspends. When the awaited operation completes, the execution state is restored and the function resumes execution. This transformation is very tedious and error-prone to perform by hand (especially when there are loops and branches), and in particular, it

preserves RAII semantics: The automatic variables created by the pre-transformed function become part of the execution state, and they are destructed at the "natural" time they would have been destructed prior to transformation.

As a result, switching to `co_await` brings us full circle back to plain old RAII. Behind the scenes, the compiler is doing the wacky transformations that we tried to mimic with `ensure_close` and `shared_close`. But `co_await` lets us write the code in a far more natural way.

[1] There's also `std::future`, but its lack of composability makes it a poor choice for asynchronous programming.

Raymond Chen

Follow